Institut Polytechnique des Sciences Avancées

Real-Time Embedded Systems

Final Assignment Report : Schedule search

Written by :

DELSARTE Néo

Monday, April 28

Academic year 2024-2025

Contents

Introduction

As part of this course on real-time embedded systems, the objective of this project is to develop a program capable of determining a feasible schedule for a given set of tasks. The first step involves verifying whether a valid schedule exists. If it does, the next step is to implement an algorithm that explores all possible task permutations in a non-preemptive manner, with the aim of identifying the most optimal scheduling solution.

This project highlights the importance of scheduling in real-time embedded systems, where meeting deadlines is crucial for the proper functioning of applications. By exploring different task permutations, we aim to minimize the total waiting time and ensure that all tasks are executed within their allotted time frames. This approach helps us understand the challenges associated with real-time scheduling and develop essential skills for designing efficient and reliable embedded systems.

I - Check schedulability

To begin the project, we first define the set of tasks involved. These are presented in the following table.

Task	**C**	$\mathbf{T_i}$
τ_1	2	10
τ_2	3	10
τ_3	2	20
τ_4	2	20
τ_5	2	40
τ_6	2	40
τ_7	3	80

Table 1: Task set with computation times (C) and periods (T_i)

Before proceeding with the implementation and searching for a suitable schedule, it is crucial to verify whether the task set is schedulable. To assess this, we refer to the concept of utilization introduced during the course. For a single task, utilization is defined as:

$$U_i = \frac{C_i}{T_i}$$

Then, we can calculate the total utilization for the entire task which is :

$$U = \sum_i \frac{C_i}{T_i}$$

This parameter provides a first indication of whether the task set is schedulable. If $U < 1$, we can conclude that the task set is schedulable, although it does not tell us how or with which scheduling protocol. On the other hand, if $U > 1$, the task set is not schedulable.

Here the calculation of the utilization gives :

$$\begin{aligned} U &= \frac{2}{10} + \frac{3}{10} + \frac{2}{20} + \frac{2}{20} + \frac{2}{40} + \frac{2}{40} + \frac{3}{80} \\ &= 0.8375 \end{aligned}$$

Based on this result, we can conclude that the task set is schedulable. We can therefore proceed to search for a suitable scheduling strategy.

II - How the Scheduling algorithm works

Hyperperiod

In this section, we will examine how the code operates and the choices made to ensure its functionality. A straightforward way to search for a valid schedule is to begin by computing the hyperperiod, since if a schedule is feasible over one hyperperiod, it will repeat correctly over time. In our case, the hyperperiod is:

$$Hyperperiod = lcm(T_1, T_2, \ldots, T_n) = 80$$

```
H = lcm(*[task["T"] for task in tasks])
print(f"Hyperperiod={H}")
```

Naive approach

Then, we can generate each repetition of every task, which we call a job. For example, if task 1 repeats every 10 units of time and the hyperperiod is 80, we will create 8 jobs for task 1.

```
jobs = []
for task in tasks:
    for t in range(0, H, task["T"]):
        jobs.append({
            "task": task["name"],
            "release": t,
            "deadline": t + task["T"],
            "execution": task["C"]
        })
```

Once this is done, we will have a total of 29 jobs. A naive approach to continue the code would be to generate every possible permutation among all 29 jobs. This results in 29! possibilities, which is more than 10^{30}. This poses a significant problem in terms of computational complexity and time. One way to reduce the number of possibilities is to avoid permutations where certain jobs of a task arrive before the previous ones. For example, if job 2 of task 1, named τ_{12}, arrives before job 1 of the same task, τ_{11}, this is not possible.

This is a good way to reduce the number of configurations to test, but I decided to use another method here.

Recursive approach

The method I chose to use is based on a recursive approach, meaning our function will call itself multiple times. This allows us to create a code that searches the schedule in a tree-like graph structure. To explain how the code works, I will only use the first three tasks to make it more understandable.

The principle of the code is that the function's arguments are "jobs," which contains all the jobs that have not yet been chosen, "current_time," which is the current time at the moment of the code execution, "scheduled," which is a list containing all the jobs that have already been chosen, and finally "total_wait," which contains the total waiting time.

To avoid generating every permutation, I use the "release" characteristic of the task, which corresponds to the time at which the task can be executed.

Each time the function is called, it searches the "jobs" list for jobs that are ready, using the condition (release <= current_time). For each job that is ready, it verifies that executing it will not lead to missing the deadline; otherwise, it skips that solution. If it does not miss the deadline, it calculates the waiting time for the task, removes the task from the "jobs" list, and adds it to the "scheduled" list. The final step is to call the same function with the new arguments as explained before.

In your code, the `.pop()` method is used to perform a backtracking operation. Here's how it works in the context of your `schedule_jobs` function:

1. **Adding a job to the `scheduled` list**: When we find a job that is ready to be executed, we add it to the `scheduled` list with `scheduled.append({**job, "start": start})`.

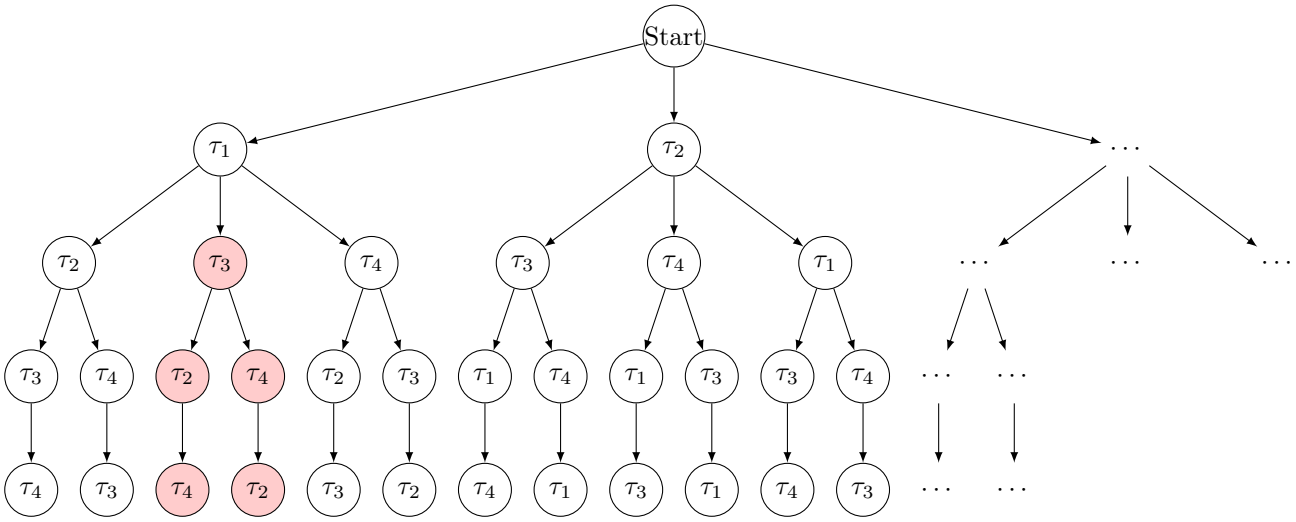
2. **Recursive call to the `schedule_jobs` function**: After adding the job to the `scheduled` list, we call the `schedule_jobs` function recursively with the updated arguments, including the updated list of remaining jobs and the current time.

3. **Removing the added job**: Once the recursive call is complete, we use `scheduled.pop()` to remove the last added job from the `scheduled` list. This allows us to revert to the previous state before the job was added, which is essential for exploring other possible permutations of the jobs.

In summary, `.pop()` allows us to backtrack and undo the last action taken, which is crucial for the backtracking process. This enables us to explore all possible combinations of jobs without retaining previous choices that do not lead to an optimal solution.

Example with a tree

To demonstrate why this code will perform faster than the example we provided earlier, I will explain using a tree graph and an example with 4 tasks.



The tree above illustrates a portion of the possible ways to choose the order of 4 tasks. For example, we could select task 1, then task 2, followed by task 3, and finally task 4. This is just one combination among many. Now, let's explain how the algorithm works. Initially, it will choose between τ_1, τ_2, τ_3, and τ_4 (assuming we have only 4 tasks). If we choose τ_1, we verify that task 1 can be executed without missing its deadline. Then, we proceed to choose among τ_2, τ_3, and τ_4. Suppose we select τ_2. The process continues with another choice until we complete the sequence. Once the sequence is complete, the code performs a backtrack. For instance, if the sequence is $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3 \rightarrow \tau_4$, the code will backtrack to $\tau_1 \rightarrow \tau_2$ and explore the other branch, $\tau_1 \rightarrow \tau_2 \rightarrow \tau_4 \rightarrow \tau_3$. This backtracking continues, exploring all possible branches of the tree.

Now, let's discuss the advantage of this method. Suppose we choose τ_1, then τ_3, but upon testing, we find that τ_3 will miss its deadline, as shown in the tree. The code will then backtrack without exploring the rest of the branch, saving time by avoiding the paths marked in red.

Here, we have considered only the permutations between 4 tasks. However, if we apply the same reasoning to scenarios with more tasks and multiple jobs per task, this method will help us avoid exploring thousands or even millions of invalid paths.

Performance Criterion

Now that we have a better understanding of how the function works, we can discuss how to choose the best schedule among all those that meet the deadlines. Our goal here is to minimize the total waiting time. This represents the time a job has to wait because another job is being executed. For example, if we execute τ_{11} first and then τ_{31}, job 1 of task 3 has to wait 2 units of time before it can be executed. In other words, it is the difference between the time when the job is released (i.e., it can be executed) and the time it starts executing. Among all the possible solutions, we will choose the one that minimizes this quantity.

III - Computational complexity

Now that we understand how the code works and how it searches for the optimal schedule, it's clear that the number of tasks can significantly increase the complexity. Task 7, with its period of 80, raises the hyperperiod to 80, which impacts the runtime and the number of branches to explore. As an anecdote, I tried running the code with all the tasks, and after more than 7 hours, I decided to stop it. The code had tested around 9 billion possibilities and still wasn't finished.

I decided to start optimizing the code to make it faster and more efficient.

First improvement

I discovered a solution that could save a significant amount of time. By testing at the beginning of the function whether the total waiting time exceeds the best total waiting time found so far, we can avoid exploring branches that already have a worse total waiting time. This additional condition, along with the missed deadline check, greatly reduces computational complexity and runtime. For example, when testing the algorithm with the first six tasks, without this improvement, the code generated a total of 82944 schedules. With the improvement, we can reduce this number to 8980 schedules, clearly demonstrating that this optimization significantly speeds up our algorithm.

Second improvement

A second improvement I discovered involves ordering the jobs by their deadlines when searching for jobs that are ready to be executed. By sorting these jobs in ascending order of their deadlines, the code prioritizes jobs with the closest deadlines first. This approach can save time because, to minimize waiting time, it's natural to execute jobs as soon as they are released.

IV - Results and Analysis

First scheduling: no task misses its deadline

The scheduling algorithm was successfully executed on the full set of seven periodic tasks. Despite the large hyperperiod and the combinatorial explosion of job sequences, the program completed its run in approximately 6-7 hours.

The best schedule found resulted in a total waiting time of **130 time units**. All job deadlines were respected, including those of task τ_5, which wasn't initially allowed to miss its deadline if necessary.

This result confirms the feasibility of finding a fully valid non-preemptive schedule for this task set, even under tight timing constraints, using exhaustive search methods when supported by sufficient computation time.

We can see below a plot of the schedule.

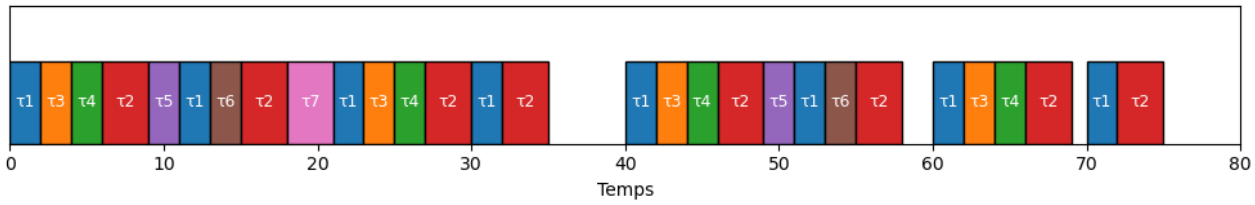


Figure 1: Optimal schedule for the first case

The following table shows the detailed execution schedule of all jobs in the optimal non-preemptive schedule. For each job, we provide the execution interval, the corresponding waiting time, and indicate any idle periods where the CPU was not utilized.

Job	Execution Interval	Waiting Time	Job	Execution Interval	Waiting Time
τ_1	[0, 2]	0	τ_3	[42, 44]	2
τ_3	[2, 4]	2	τ_4	[44, 46]	4
τ_4	[4, 6]	4	τ_2	[46, 49]	6
τ_2	[6, 9]	6	τ_5	[49, 51]	9
τ_5	[9, 11]	9	τ_1	[51, 53]	1
τ_1	[11, 13]	1	τ_6	[53, 55]	13
τ_6	[13, 15]	13	τ_2	[55, 58]	5
τ_2	[15, 18]	5	idle	[58, 60]	-
τ_7	[18, 21]	18	τ_1	[60, 62]	0
τ_1	[21, 23]	1	τ_3	[62, 64]	2
τ_3	[23, 25]	3	τ_4	[64, 66]	4
τ_4	[25, 27]	5	τ_2	[66, 69]	6
τ_2	[27, 30]	7	idle	[69, 70]	-
τ_1	[30, 32]	0	τ_1	[70, 72]	0
τ_2	[32, 35]	2	τ_2	[72, 75]	2
idle	[35, 40]	-	idle	[75, 80]	-
τ_1	[40, 42]	0	-	-	-

Total waiting time: 130 time units.

The fact that the algorithm found a solution where all tasks meet their deadlines, including τ_5, demonstrates the capability of exhaustive search methods to discover non-obvious yet valid solutions without the need for deadline relaxation.

Second scheduling: τ_5 is allowed to miss a deadline

In this scenario, the scheduling algorithm is re-executed with a relaxed constraint: task τ_5 is now allowed to miss its deadline once if necessary. This relaxation introduces greater flexibility in the job ordering, potentially leading to a more efficient overall schedule in terms of total waiting time.

The motivation behind this test is that τ_5, having a relatively long period and likely lower criticality, could reasonably be deprioritized to favor tighter deadlines from other tasks. By allowing this exception, we aim to observe whether the algorithm can significantly reduce total waiting time.

After running the complete search again with this new configuration, the algorithm produced a new optimal schedule. Compared to the first case, we note the following:

- Task τ_5 does miss a deadline in the selected schedule,
- The schedule is exactly the same as the first case
- The total waiting time is then the same

We can see the new schedule in the figure below.

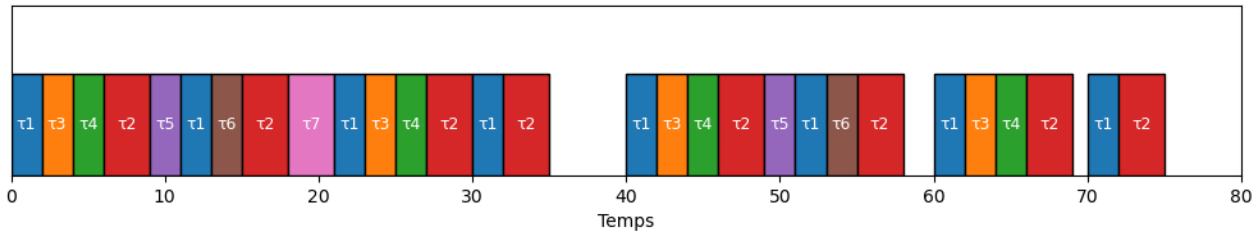


Figure 2: Optimal schedule when τ_5 is allowed to miss its deadline

The following table shows the detailed execution schedule for this configuration. Idle periods are indicated as before.

Job	Execution Interval	Waiting Time	Job	Execution Interval	Waiting Time
τ_1	[0, 2]	0	τ_3	[42, 44]	2
τ_3	[2, 4]	2	τ_4	[44, 46]	4
τ_4	[4, 6]	4	τ_2	[46, 49]	6
τ_2	[6, 9]	6	τ_5	[49, 51]	9
τ_5	[9, 11]	9	τ_1	[51, 53]	1
τ_1	[11, 13]	1	τ_6	[53, 55]	13
τ_6	[13, 15]	13	τ_2	[55, 58]	5
τ_2	[15, 18]	5	idle	[58, 60]	-
τ_7	[18, 21]	18	τ_1	[60, 62]	0
τ_1	[21, 23]	1	τ_3	[62, 64]	2
τ_3	[23, 25]	3	τ_4	[64, 66]	4
τ_4	[25, 27]	5	τ_2	[66, 69]	6
τ_2	[27, 30]	7	idle	[69, 70]	-
τ_1	[30, 32]	0	τ_1	[70, 72]	0
τ_2	[32, 35]	2	τ_2	[72, 75]	2
idle	[35, 40]	-	idle	[75, 80]	-
τ_1	[40, 42]	0	-	-	-

Total waiting time: 130 time units.

This result shows that allowing task T5 to miss its deadline has no effect on the system's behavior or outcome. It underscores that, in some cases, relaxing timing constraints for non-critical tasks can be a safe and efficient design choice, rather than a trade-off.

Conclusion

This project on real-time embedded systems aimed to develop a program capable of determining a feasible schedule for a given set of tasks. After verifying the feasibility of the tasks, an algorithm was implemented to explore all possible task permutations in a non-preemptive manner, with the goal of identifying the most optimal scheduling solution.

The results show that the algorithm successfully found a schedule where no task missed its deadline, with a total waiting time of 130 time units. Even when allowing one task to miss its deadline, the optimal schedule remained the same, highlighting the robustness of the approach used.

Improvements made to the algorithm, such as checking the total waiting time and sorting tasks by their deadlines, significantly reduced computational complexity and execution time. These optimizations demonstrate the importance of efficient techniques for real-time scheduling, which are crucial for the proper functioning of embedded systems.

In conclusion, this project provided a better understanding of the challenges associated with real-time scheduling and helped develop essential skills for designing efficient and reliable embedded systems.

List of Figures